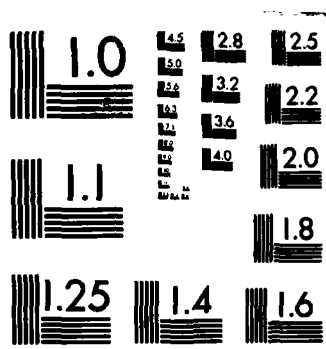
AD-A154 341   VAX TRICOMP USER'S GUIDE(U) NAVAL SURFACE WEAPONS   1/1
CENTER DAHLGREN VA   J J ZALOUDEK JUL 84 NSWC/TR-84-97
SBI-AD-F350 022

UNCLASSIFIED   F/G 9/2   NL

END
FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A154 341

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NSWC TR 84-97	2. GOVT ACCESSION NO. AD-A154341	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) VAX TRICOMP USER'S GUIDE		5. TYPE OF REPORT & PERIOD COVERED Final
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John J. Zaloudek		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Surface Weapons Center (Code K53) Dahlgren, VA 22448		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 36801
11. CONTROLLING OFFICE NAME AND ADDRESS Strategic Systems Program Office Washington, DC 20376		12. REPORT DATE July 1984
		13. NUMBER OF PAGES 61
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

TRIDENT Higher Level Language
THLL
VAX 11/780
Compiler
program
implementation
runtime system

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The TRIDENT Higher Level Language (THLL) is implemented on the VAX 11/780 based TRIDENT Software Generation System (SGS) for three target machines: the TRIDENT Basic Processor (BP), the Motorola MC68000, and the VAX 11/780. This document describes implementation features of THLL, which are unique for the VAX, and the runtime support system for the VAX.

FOREWORD

This document was written in the Operational Support Branch (K53), Submarine Launched Ballistic Missile (SLBM) Software Development Division (K50), of the Strategic Systems Department (K) at the Naval Surface Weapons Center (NSWC), Dahlgren, Virginia.

The purpose of this document is to describe implementation features of the TRIDENT Higher Level Language (THLL), which are unique for the VAX 11/780, and the runtime support system for THLL on the VAX. It supplements the THLL Reference Manual for programs designed to run on the VAX.

This report was extensively reviewed by personnel in the Operational Support Software Systems Group in K53, the Quality Assurance Branch (K52), and the Operational Systems Branch (K54).

This project was funded by the Strategic Systems Program Office, Washington, DC 20376, under task number 36801.

Questions, comments, and suggestions concerning the material presented in this document should be directed to the Commander, Naval Surface Weapons Center, ATTN: K53, Dahlgren, Virginia 22448.

Approved by:

THOMAS A. CLARE, Head
Strategic Systems Department

CONTENTS

CHAPTER 1

INTRODUCTION

VAX TRICOMP is a compiler for the TRIDENT Higher Level Language (THLL) that runs on the VAX 11/780 and produces code for the VAX. This document describes implementation features of THLL, which are unique for the VAX, and the runtime support system for the VAX.

Differences between the VAX compiler and other THLL compilers are due to differences in the hardware of the target machines and differences in the software runtime environment of the target machines. This document is not self-contained. It is to be used in combination with the THLL Reference Manual (Reference 1). Details about the VAX architecture can be found in the VAX Architecture Handbook (Reference 2).

VAX TRICOMP allows the use of THLL as a general purpose programming language for the VAX 11/780 computer. THLL is compatible with the other languages that are available on the VAX.

The VAX architecture views data as 8-bit bytes, 16-bit words, 32-bit longwords, and 64-bit quadwords. The address of the data is the byte address of the first byte of the data. THLL views data in 32-bit units called THLL words (which are VAX longwords). Throughout this User's Guide, the term "word" refers to a 32-bit THLL word and is not to be confused with the VAX 16-bit word. Also, a THLL doubleword (or simply doubleword) is a VAX quadword.

CHAPTER 2

BASIC ELEMENTS OF THLL

2.1 THLL CHARACTER SET

The THLL character set consists of the following ASCII subset:

Uppercase letters A-Z

Numerals 0-9

The following special characters:

Character	Name	Character	Name
	Space	.	Period
!	Exclamation point	/	Slash
"	Quotation mark	:	Colon
#	Number sign	;	Semicolon
$	Dollar sign	<>	Angle brackets
%	Percent sign	=	Equal sign
&	Ampersand	?	Question mark
'	Apostrophe	@	At sign
()	Parentheses	[]	Square brackets
*	Asterisk	\	Backslash
+	Plus sign	^	Circumflex
,	Comma	_	Underline
-	Minus sign		

Not included in the THLL character set are the lowercase letters (codes X'60' - X'7E') and the nonprintable characters (codes X'00' - X'1F' and X'7E'). When these characters are used in a THLL source file they are treated as follows:

A. Lowercase letters are converted to uppercase letters (codes X'40' - X'5E') except when they appear in THLL strings or in comments or remarks.

B. Nonprintable characters are converted to question marks (?).

The listing file produced by VAX TRICOMP reflects these character conversions.

A table of ASCII characters is included in Appendix A.

2.2 OPERATORS

The value of LOC X is the virtual address of the THLL word or doubleword containing X. A virtual address is a 32-bit quantity of type POINTER.

The value of LOCA X is also the virtual address of the THLL word or doubleword containing X. In this case, the type of the value is INTEGER. The bit patterns of LOC X and LOCA X are identical.

2.3 DELIMITERS

All of the THLL delimiters are implemented in VAX TRICOMP, but some have little or no meaning in the environment of the VAX. Examples are: EXEC, HALF, INTERRUPT, LINK.

2.4 IDENTIFIERS

The first 31 characters of identifiers are significant. The first character of the identifier must be a letter. The remaining characters can be letters, numerals, or special characters.

VAX TRICOMP accepts dot (.), underscore (_) and dollar ($) as special characters within identifiers. They can occur anywhere after the first character of the identifier (including the last character). The dollar character should not be used in user-defined globals. On the VAX, it is reserved for use in system symbol names. The dollar character is accepted by VAX TRICOMP to allow direct THLL access to the system procedures.

Care must be taken not to choose global or external names that have other meaning to the MACRO assembler. Examples are: AP, SP, FP, PC, and R0 thru R12.

2.5 CONSTANTS

For a pictorial representation of the layout of the following constants, see Section 3.3.

2.5.1 Integer and Double Constants

In VAX TRICOMP, any decimal integer that exceeds 31 bits of significance is considered to be a double value. Binary, octal, and hexadecimal integers are considered to be double when they exceed 32 bits of significance.

An integer constant is kept in one THLL word (32-bit VAX longword). A double constant is kept in a THLL doubleword (64-bit VAX quadword). VAX TRICOMP uses the normal VAX representation for these double constants.

2.5.2 Real Constants

VAX TRICOMP uses the 64-bit VAX D-floating format for THLL real numbers (see Reference 2). The range for real constants on the VAX is approximately .29E-38 through 1.7E+38. The precision for real constants on the VAX is approximately 16 decimal digits.

2.5.3 Pointer Constants

The only legal pointer constant is 0. Pointer expressions built up using THLL operators are valid at preset time and runtime. A pointer contains the byte address of the designated item. The virtual address space of the VAX is very large, and VAX TRICOMP uses it. Pointer components on the VAX should be 32 bits wide.

2.5.4 Boolean Constants

TRUE is defined to be a 32-bit integer that has all 32 bits set; however, any non-zero value tests as true. FALSE is represented as X'00000000'.

2.5.5 Strings

A string is represented by a header word followed by a block of words holding the characters of the string in standard VAX format. The characters are ASCII as defined for the VAX. Each character occupies one 8-bit byte. The characters are packed from right to left within a THLL word. This is the packing order used by all existing VAX products.

THLL strings and ALPHA variables do not have the additional descriptor used by the other VAX languages. The descriptor can be easily developed if such a quantity must be passed to a module written in a different language. Such a need is rare and does not warrant the additional overhead of having the descriptor. See Appendix C, Section 8 of the VAX Architecture Handbook

(Reference 2) for more information about VAX descriptors.

CHAPTER 3

DATA DECLARATIONS

3.1 TYPES

There are six types for classifying constants, variables, and procedures in THLL. A type may be thought of as a class of values. In general, the user should only have to know about the abstract properties of a type, the rules for the use of data types by operators and procedures, and the rules for type conversion. For portability reasons, the user should make a conscious effort to not take advantage of a particular implementation of types. In some cases, however, it may be necessary to know about the internal representation of the various types of values in memory.

Constants are assigned types as specified in Sections 2.5.1 through 2.5.5. Symbol types are specified in the corresponding declarations. Valueless items, such as statements, are assigned no type (N).

3.2 BIT NUMBER CONVENTIONS

THLL bit numbers run from 0 for the leftmost (most significant) bit to 31 for the rightmost (least significant) bit of a THLL word. For a doubleword the bit numbers run from 0 to 63.

This convention is used for defining the meaning of a component "start bit" and "number of bits." Functions such as TEST.BIT, CLR.BIT, SET.BIT, TGL.BIT, and FIND.BIT use the THLL bit number as an input parameter, and FIND.BIT can also output a THLL bit number.

THLL bit numbers are opposite of the VAX view of bit numbers as used by other languages on the VAX. The VAX bit numbering convention may need to be used when interfacing with other languages.

3.3 STORAGE FORMAT

The types INTEGER (I), DOUBLE (D), and REAL (R) have a fixed format as to number of bits, location of sign bit, and number of words required for storage.

The type HALF (H) is equivalent to the type INTEGER for all data. Type INTEGER requires one word, 32 bits.

0	1 31	bit #
S	value	word 1

S = sign bit

Type DOUBLE requires two words, 64 bits.

0 1 31	bit #
low order value	word 1

32	63	bit #
S	high order value	word 2

Type REAL requires two words, 64 bits, separated into exponent and mantissa fields. The exponent is an excess-128 value, and the mantissa is a signed magnitude value with the redundant most significant bit not represented. For more information on the VAX D-format floating point data type used for THLL, see the VAX Architecture Handbook (Reference 2).

0 15	16	17 24	25 31	bit #
mantissa part 2	S	exponent	mant 1	word 1

32 47	48 63	bit #
mantissa part 4	mantissa part 3	word 2

Type ALPHA is stored as four characters per word plus one header word. Thus, the number of words required for a string of n characters is ((n+3)/4) + 1. However, the programmer does not need to allow for the header when specifying string length.

0 7	8 15	16 23	24 31	bit #
Char. #4	Char. #3	Char. #2	Char. #1	

Eight characters of a string can also be stored in a variable of type DOUBLE.

0 7	8 15	16 23	24 31	bit #
Char. #4	Char. #3	Char. #2	Char. #1	word 1

32 39	40 47	48 55	56 63	bit #
Char. #8	Char. #7	Char. #6	Char. #5	word 2

Type POINTER requires one word, 32 bits.

0 31	bit #
virtual address	word 1

3.4 ALLOCATION OF STORAGE

Storage is allocated in the order in which data are declared. All THLL constants and data structures are allocated at THLL word or doubleword addresses. A THLL word starts at a byte address that is a multiple of 4, and a doubleword starts at a byte address that is a multiple of 8. Arrays, stacks, commons and quantities that are DOUBLE or REAL start at doubleword addresses. All other data items start at word addresses.

If in allocating a doubleword address for OWN or shared data structures, a "hole" arises, the hole remains. No attempt is made to fill the hole. Holes that arise in the constant area are used for subsequent singleword constants.

The allocation of storage within a common is the same as the allocation of storage outside the common.

The constant area, the OWN area, and the runtime stack all begin at doubleword addresses. This guarantees that REAL and DOUBLE quantities as well as arrays, stacks, and commons start at doubleword addresses. This allows the VAX hardware to operate more efficiently.

FORTRAN arrays may or may not have headers, and when they do, the header can be different from THLL array headers. As a rule, an array name should not be made EXTERNAL or passed as an argument to a procedure across languages, for example, between FORTRAN and THLL. A pointer to the origin of the data in the array should be used instead.

3.5 COMPONENTS

Even though the addressing on the VAX is in units of bytes, THLL components are based on the concept of THLL words. The offset part of the component declaration is in units of THLL words. When an integer expression is added to the pointer in a component reference, it is also considered to be in units of THLL words. VAX TRICOMP automatically multiplies these values by 4 to access the correct byte address.

The virtual address space of the VAX is quite large, and VAX TRICOMP uses it. An item on the runtime stack has a very high virtual address. POINTER components on the VAX should be 32 bits wide.

REAL and DOUBLE components reference THLL doublewords. However, they do not need to be aligned at doubleword addresses. The offset in the component declaration is in units of THLL words. Thus, a DOUBLE or REAL component occupies two offset values.

3.5.1 Indexed Components

Full word INTEGER and POINTER components index in units of THLL words, and DOUBLE and REAL components index in units of THLL doublewords.

Partial word indexed components run with increasing index from left to right. For portability reasons, characters in ALPHA or DOUBLE variables should not be accessed with INTEGER indexed components. Instead, ALPHA components should be used.

3.5.2 Alpha Components

To be compatible with most other VAX products, characters are organized from right to left. This differs from all other THLL target computers. The ALPHA component provides a machine-independent method of accessing character data. In effect, it is an indexed INTEGER component that indexes in the same direction as character data on the target computer. The type of the value of an alpha component is INTEGER, not ALPHA!

An ALPHA component variable must always be used as an indexed component variable. The OFFSET information in the component declaration is in units of THLL words. A particular character within the THLL word can be accessed by

using an appropriate character number as in the example below.

```
ALPHA COMPONENT CHR ;
OFFSET 0 FOR CHR ;     /* OPTIONAL - 0 IF NOT SPECIFIED */
                  or
ALPHA COMPONENT CHR(OFFSET 0) ;

usage:  CHR(P,I)       /* P - POINTER TO FIRST WORD THAT
                              CONTAINS CHARACTER DATA */
                       /* I - CHARACTER NUMBER */
```

Note that the following two declarations are NOT equivalent in VAX TRICOMP:

```
ALPHA COMPONENT CHAR (OFFSET 0) ;
INTEGER COMPONENT BYTE (FIELD(0,8),OFFSET 0) ;
```

The two component variables CHAR(P,I) and BYTE(P,I) may not have the same value on the VAX. The second method is not recommended.

3.6 PRESETS

The evaluation of compile time expressions is done by actually executing the VAX operations and not by simulating the arithmetic of the VAX. This allows VAX TRICOMP to perform more kinds of compile time expression evaluations. For example, simple real operations (+, -, etc.) and shift operations are supported.

Compile time expressions evaluated by VAX TRICOMP are the same as runtime expression evaluation on the VAX. Presets using indexed components work the same as runtime indexed components.

The preset functions LINKWORD(LOC P,N) and INITWORD(LOC P,N) each generate a THLL word of type INTEGER containing the address of procedure P. The N parameter of these functions is accepted by VAX TRICOMP for compatibility with other THLL compilers, but it is not used in the preset.

CHAPTER 4

EXPRESSIONS AND STATEMENTS

4.1 EXPRESSION TYPES

Pointers contain addresses in units of bytes. THLL considers memory to be in units of 32-bit THLL words. Offsets from a pointer value are in units of THLL words. Whenever an integer expression is added to or subtracted from a pointer, VAX TRICOMP automatically multiplies the integer value by four. When the difference of two pointer values is desired, VAX TRICOMP divides the integer result by four.

There are no expressions of type HALF. Variables declared to be type HALF are considered to be type INTEGER by VAX TRICOMP.

4.2 PROCEDURES

A THLL procedure can be called from another THLL procedure, a FORTRAN program or subroutine, a MACRO program, or the operating system. A THLL procedure can call itself, other THLL procedures, FORTRAN subroutines, and MACRO programs. VAX TRICOMP uses the standard procedure calling conventions used by all languages for the VAX. Therefore, THLL procedures may be compatible with other languages.

THLL allows recursion. THLL procedures running on the VAX can be executed in a recursive manner. Variables sensitive to each invocation of a procedure must be shared (non-OWN).

A runtime exception (interrupt) occurs when a procedure references a missing parameter passed by reference. This can happen when required actual parameters are missing in the procedure call.

Appendix B contains details about the runtime environment on the VAX. Appendix C shows a method for preparing programs for the VAX.

4.2.1 LINK and EXEC Procedures

LINK and EXEC procedures are assumed to be GLOBAL. It is not considered an error if they are also declared to be GLOBAL.

Any global procedure can be used as the entry point for executing the object program. The MAIN directive is used to specify the execution entry point.

4.2.2 VAX Argument Transmission Modes

The VAX procedure calling conventions define three mechanisms for transmitting parameters to subroutines (procedures): by value, by reference, and by descriptor. As far as other languages are concerned, the normal THLL mechanism for passing parameters is by VAX reference. This applies to all THLL data including strings and ALPHA variables. Even though the VAX reference mechanism is the normal mode of transmitting THLL parameters, THLL procedures can pass parameters via the value and descriptor mechanisms.

VAX TRICOMP provides the inline function VAL$ to pass a parameter by the VAX value mechanism. The VAL$(exp) function is valid only when applied to an actual parameter. Depending upon the type of exp, VAL$(exp) yields the following results:

A. INTEGER and POINTER - The THLL word is passed to the called procedure.

B. DOUBLE - The least significant word of the THLL doubleword is passed to the called procedure.

C. REAL - An F-format real is passed to the called procedure. See Reference 2 for a description of an F-format floating point real.

The VAL$ function is intended to be used only when calling procedures written in some language other than THLL. If it is used when calling a THLL procedure, the called procedure will not access the data properly. The following example uses the VAL$ function to pass a parameter by the VAX value mechanism.

```
EXTERNAL PROCEDURE LIB$SIGNAL(INTEGER VALUE) ;
         .
         .
         .
LIB$SIGNAL(VAL$(SS$_ABORT)) ;
```

A combination of the VAX reference mechanism and components must be used to pass a parameter by the descriptor mechanism. In this approach, the descriptor is built up in a block of memory using components, and the first

word of the memory block is passed by THLL reference (as opposed to passed by THLL value) to the called procedure. Appendix C, Section 8 of the VAX Architecture Handbook (Reference 2) defines all VAX Descriptors. The following example shows passing a parameter by the descriptor mechanism.

```
COMMENT The procedure OTS$CVT_TO_L converts an octal string to its
        binary representation.  The parameters to OTS$CVT_TO_L are:
        1. The string to be converted, passed by descriptor
        2. The converted value, passed by reference
        3. The number of bytes the converted value occupies, passed by
           value
        4. A flag that if it is 1, then ignore blanks in the string,
           passed by value.

        The call to OTS$CVT_TO_L converts a string that is contained
        in a THLL ALPHA variable. ;

EXTERNAL INTEGER PROCEDURE OTS$CVT_TO_L(DOUBLE,INTEGER,INTEGER VALUE,
 INTEGER VALUE) ;

INTEGER COMPONENT DSC.CLASS(FIELD(0,8), OFFSET 0) ;
INTEGER COMPONENT DSC.TYPE(FIELD(8,8), OFFSET 0) ;
INTEGER COMPONENT DSC.LEN(FIELD(16,16), OFFSET 0) ;
POINTER COMPONENT DSC.PSTR(OFFSET 1) ;

OWN INTEGER RESULT,STATUS ;
OWN ALPHA STRING(4) ;
OWN DOUBLE STR_DESC ;

PRESET
     BEGIN
     STRING = #'10253' ;
     DSC.CLASS(LOC STR_DESC) = 1 ;
     DSC.TYPE(LOC STR_DESC) = 14 ;
     DSC.LEN(LOC STR_DESC) = 5 ;
     DSC.PSTR(LOC STR_DESC) = LOC STRING+1 ;
     END ;
.
.
.
STATUS = OTS$CVT_TO_L(STR_DESC,RESULT,VAL$(4),VAL$(1)) ;
```

4.2.3 Optional Arguments

The THLL functions, ARGTYPE and ARGSYNCL, use an argument descriptor list. In order for this list to be formed, OPTARG must be specified in the parameter list of the procedure that uses these functions. Likewise, if this procedure is called from another compile unit, the external declaration for

this procedure in that compile unit must also contain the OPTARG specification.

ARGSYNCL is used to determine the syntactic class of the I'th argument of a call to a procedure using optional arguments. More information about ARGSYNCL may be found in the THLL Reference Manual (Reference 1). For the VAX, the meaning of the integer value returned by ARGSYNCL is:

Argument Class	Value Returned
one-dimensional array	1
two-dimensional array	2
three-dimensional array	3
simple variable	4
stack	5
procedure	6
device	7
format	8

CHAPTER 5

THLL I/O ON THE VAX

The THLL Reference Manual (Reference 1) contains the specification for the various THLL Input/Output (I/O) functions. This chapter describes the peculiarities of the THLL I/O implementation available on the VAX.

5.1 THLL FILE CONTROL BLOCK

The THLL runtime system on the VAX maintains a THLL File Control Block (THLLFCB) for each file in use. It is allocated when a file is opened and released when the file is closed. The THLLFCB is used to identify the file via the pointer to the THLLFCB returned by OPEN. In some cases, noted below, status indicators for the file are returned to the user in the THLLFCB.

5.2 OPEN PROCEDURE

The filename parameter in the call to OPEN is used to determine the file being opened. In this manner, one device is not tied to one file. The filenames '' (null string), 'INPUT', and 'OUTPUT' have special meaning for sequential coded files. These names are illegal for random binary files. The file type DAT is appended to the filename if the user does not specify a file type in the logical name table.

A null string is assigned the filename INPUT for the KBDSS and OUTPUT for CPRINT or SPRINT. Filename INPUT reads from SYS$INPUT and filename OUTPUT writes to SYS$OUTPUT.

Sequential output files are tagged FORTRAN print control files, i.e., the first character of each record is used as a carriage control character if the file is routed to a print device. If the first character of a record is not a space, special handling is required before the file can be routed to a printer. See the VAX-11 FORTRAN Language Reference Manual (Reference 3).

The following table shows the file usage for filenames:

	''	'INPUT'	'OUTPUT'	'filename'
KBDSS	input	input	output	input
CPRINT	output	error	output	output
SPRINT	output	error	output	output
MDF	error	error	error	input/output
MTF	error	error	error	input/output
ICL	error	error	error	input/output

Note: INPUT reads from SYS$INPUT and OUTPUT writes to SYS$OUTPUT.

5.3 CLOSE PROCEDURE

Files must be closed. If an output file is not closed, some data may be lost when the THLL program terminates.

5.4 REWIND FUNCTION

A rewind function is available for sequential files. To use it, the following external declaration must be made:

```
EXTERNAL PROCEDURE THL$REWIND(POINTER VALUE) ;
```

The form of the procedure call is:

```
THL$REWIND(P)
```

where

P - pointer to a THLLFCB as returned by a call to the OPEN procedure.

5.5 SPECIAL LIMITATIONS

The following limitations apply to sequential coded files:

A. The size of a formatted line is four times the size given in the OPEN call. Thus a size of 20 should be used to read 80 character input lines, and a size of 33 should be used for 132 column printed output. The maximum size for an input or output line is 132 characters. No distinction is made between uppercase and lowercase letters.

B. If an end-of-file is encountered during a read, then the fourth word of the THLLFCB is set to 1. If the N'th item in the expanded format contains an improper value, the fourth word of the THLLFCB is set to -N. It is set to 0 if there are no format errors.

C. The maximum field width for D, O, H, I, F, and E format is 32 characters. A doubleword output via D format can be up to 64 characters.

D. The minimum F format is F'3' (sX.).

E. The minimum E format is E'9' (s0.XEsXXX).

F. Blank input fields are treated as 0.

G. R'0' skips to the next page and prints a blank line when writing. It is ignored when reading.

H. Format repeats can be nested 10 deep.

The following limitations apply to random binary files:

A. The file position must be greater than 0.

B. An attempt to read a nonexistent record results in a buffer filled with 0.

CHAPTER 6

STANDARD PROCEDURES

All standard procedures described in the THLL Reference Manual (Reference 1) are available on the VAX.

Most of the standard functions and a number of operators such as ** are implemented as routines in the THLL runtime library, THLLIB. Appendix C shows how to access this library.

The following sections describe VAX machine dependencies for the standard procedures.

6.1 FIXH, FIXI, AND FIXD FUNCTIONS

The value returned is the integer portion of the argument. Thus FIXI(-4.5) = -4 and FIXI(4.5) = 4. This is also the case when the compiler automatically generates a REAL to INTEGER type conversion.

6.2 MATHEMATICAL FUNCTIONS

The mathematical routines such as the trigonometric functions, SQRT, LN, EXP, etc. are in the VAX system libraries. These libraries are automatically searched by the VAX loader. No check is made to determine if the argument for a trigonometric function is in the appropriate THLL range. The response by these routines to illegal parameter values depends on the routines in the system libraries. They do not necessarily respond the same as described in the THLL Reference Manual.

6.3 SHIFT FUNCTIONS

A HALF operand is a 32-bit INTEGER quantity and is shifted as such.

6.4 SWA FUNCTION

The SWA function is used only as a means of changing the type of a number from type INTEGER to type POINTER.

CHAPTER 7

UTILITY PROCEDURES

Each utility procedure that is used must be declared external. The form for that external declaration appears with each procedure. The effect, restrictions, and applications are briefly defined. These procedures can be found in the THLL runtime library, THLLIB. Appendix C shows how to access this library.

7.1 CONVALPH

CONVALPH is an integer procedure which converts a substring of an ALPHA variable to an integer. This integer is returned as the value of the procedure unless the substring is not convertible. A substring is convertible if it contains only characters 0 through 9. The procedure allows the THLL programmer to designate the character position P of the ALPHA string on which to start the conversion and the length L of the ALPHA string to be converted. If the position is not specified, then 0 is assumed. If the length is not specified, then the current length of the entire ALPHA string minus P is assumed. If P+L exceeds the current length of the ALPHA variable, the string is not convertible.

The form of the procedure call is:

```
CONVALPH(NAME[,P[,L]])
```

where

NAME - the identifier of the ALPHA string or a pointer to the ALPHA.

P - the start character position within the ALPHA at which to begin conversion (first position is 0).

L - the number of characters to be converted.

The returned value of the procedure is:

Returned Value	Condition
desired integer	the ALPHA is convertible
-1	the ALPHA is not convertible

The external declaration is:

EXTERNAL INTEGER PROCEDURE CONVALPH(ALPHA,OPTARG) ;

For details on *OPTARG* see Reference 1.

Examples of CONVALPH:

A. If the declaration is:

ALPHA NAME(13) ;

and the assignment is:

NAME=#/NUMBER IS 1234/ ;

then CONVALPH(NAME,10,4) returns 1234.

B. If the declaration is:

ALPHA NUMB(3) ;

and the assignment is:

NUMB=#/1234/ ;

then CONVALPH(NUMB) returns 1234.

C. If the declaration is:

ALPHA MESSAGE(14) ;

and the assignment is:

MESSAGE=#/HEX VALUE IS D8/ ;

then CONVALPH(MESSAGE,13,2) returns -1 because D8 is not convertible.

D. If the declaration is:

```
ALPHA B(3) ;
```

and the assignment is:

```
B=#/-250/ ;
```

then CONVALPH(B) returns -1 because the negative sign is not convertible.

E. If the declaration is:

```
ALPHA B(15) ;
```

and the assignment is:

```
B=#/SPEED IS 250 MPH/ ;
```

then CONVALPH(B,9,3) returns 250.

7.2 CONVINT

CONVINT is a procedure which converts an integer INT into an ALPHA string of a specified length L and substitutes it into a given ALPHA variable starting at a specified position P. Only non-negative integers are convertible. If the position P is not specified, then 0 is assumed. If the length is not specified, then the current length of the ALPHA variable minus P is assumed. If P+L exceeds the current length of the ALPHA variable, the number is not convertible.

The string produced from INT is defined as follows:

If INT < 0 or INT ≥ 10**L, then the string is a sequence of L asterisks:

```
#/*...*/
```

Otherwise, the string is a sequence of L characters consisting of the right-justified decimal digits resulting from the conversion. If necessary, the field is padded with leading zeros.

```
#/0 ... 0 dk ... d1 d0/,
```

The di are the decimal digits of the number:

```
INT = d0 + d1 * 10 + ... + dk * 10**k
```

The form of the procedure call is:

CONVINT(INT,NAME[,P[,L]])

where

INT - the integer to be converted.

NAME - the identifier of the ALPHA variable or a pointer to the ALPHA.

P - the start character position within the ALPHA.

L - the number of characters into which the integer is to be converted.

The returned value of the procedure is:

Returned Value	Condition
0	the integer is not convertible or the string produced is #/*...*/
-1	the integer is convertible

The external declaration is:

EXTERNAL INTEGER PROCEDURE CONVINT(INTEGER VALUE,ALPHA,OPTARG) ;

For details on OPTARG see Reference 1.

Examples of CONVINT: (b indicates a blank)

A. If the declarations are:

```
INTEGER INT ;
ALPHA NOTE(8) ;
```

and the assignments are:

```
INT = 20 ;
NOTE = #/MSLbNObbb/ ;
```

then CONVINT(INT,NOTE,7,2) produces NOTE = #/MSLbNOb20/

B. If the declarations are:

```
INTEGER INT ;
ALPHA NOTE(5) ;
```

and the assignments are:

```
INT = 10 ;
NOTE = #/bbbbbb/ ;
```

then CONVINT(INT,NOTE) produces NOTE = #/000010/

C. If the declarations are:

```
INTEGER INT ;
ALPHA N(8) ;
```

and the assignments are:

```
INT = -250 ;
N = #/bbbbbbbbb/ ;
```

then CONVINT(INT,N) produces N = #/*********/

D. If the declarations are:

```
INTEGER INT ;
ALPHA M(8) ;
```

and the assignments are:

```
INT = 500 ;
M = #/bb/ ;
```

then CONVINT(INT,M) produces M = #/**/

7.3 EPRINT

EPRINT prints a line on file SYS$OUTPUT. The form of the procedure call is:

```
EPRINT(MSG)
```

where

MSG - the ALPHA string to be printed or a pointer to the ALPHA.

The MSG is printed on file SYS$OUTPUT in the following format:

```
**** ERROR AT LABEL <string>
```

The external declaration is:

```
EXTERNAL PROCEDURE EPRINT(ALPHA) ;
```

7.4 TERM

TERM prints a line on file SYS$OUTPUT. The form of the procedure call is:

```
TERM(MSG)
```

where

MSG - the ALPHA string to be printed or a pointer to the ALPHA.

The external declaration is:

```
EXTERNAL PROCEDURE TERM(ALPHA) ;
```

7.5 THLLERR

THLLERR prints a line on file SYS$OUTPUT. THLLERR also causes the program to abort with a traceback.

The form of the procedure call is:

```
THLLERR(MSG,NUM)
```

where

MSG - the ALPHA string to be printed or a pointer to the ALPHA.

NUM - an integer number to be printed in hexadecimal format.

THLLERR causes the following line to be printed on file SYS$OUTPUT and then the program is aborted with a traceback.

```
<string> AT LOCATION <number>
```

The external declaration is:

```
EXTERNAL PROCEDURE THLLERR(ALPHA,INTEGER VALUE) ;
```

7.6 TRUNC

TRUNC is an integer procedure that can be used to obtain the singleword representation of the integer portion of a real expression. For example, TRUNC(2.5) is 2, and TRUNC(-2.5) is -2. This is the same as the FIX functions described in Section 6.1.

The form of the procedure call is:

```
TRUNC(R)
```

where

R - any real valued expression

It should be noted that no attempt is made to determine whether the value of the integer obtained can fit into a singleword.

The external declaration is:

```
EXTERNAL INTEGER PROCEDURE TRUNC(REAL VALUE) ;
```

7.7 TRUNCD

TRUNCD is a double procedure that can be used to obtain the doubleword representation of the integer portion of a real expression.

The form of the procedure call is:

```
TRUNCD(R)
```

where

R - any real valued expression

The external declaration is:

```
EXTERNAL DOUBLE PROCEDURE TRUNCD(REAL VALUE) ;
```

CHAPTER 8

REFERENCES

1. Hartmut G. Huber, THLL Reference Manual, NSWC TR 84-101, Jul 1984.

2. VAX Architecture Handbook, Digital Equipment Corporation, 1981.

3. VAX-11 FORTRAN Language Reference Manual, Digital Equipment Corporation, Apr 1982.

4. VAX-11 MACRO Reference Manual, Digital Equipment Corporation, May 1982.

5. VAX-11 Linker Reference Manual, Digital Equipment Corporation, May 1982.

APPENDIX A

ASCII CHARACTER SET

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	¢	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	¦
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation	EM	End of Medium
LF	Line Feed	SUB	Substitute
VT	Vertical Tab	ESC	Escape
FF	Form Feed	FS	File Separator
CR	Carriage Return	GS	Group Separator
SO	Shift Out	RS	Record Separator
SI	Shift In	US	Unit Separator
SP	Space	DEL	Delete

The table above represents the ASCII character set. At the top of the table are hexadecimal digits (0 to 7), and to the left of the table are hexadecimal digits (0 to F). To determine the hexadecimal value of an ASCII

character, use the hexadecimal digit that corresponds to the row in the "units" position, and use the hexadecimal digit that corresponds to the column in the "16's" position. For example, the value of the character representing the equal sign is 3D.

APPENDIX B

RUNTIME ENVIRONMENT ON THE VAX

B.1 CONTROL SECTIONS

THLL programs are broken into four control sections on the VAX. Control sections are used to identify characteristics associated with a data area or instruction area of a program. The following table defines the VAX control sections. It also indicates the memory alignment and protections associated with each section.

Section	Purpose	Alignment	Sharable	Protections
O$$	OWN data area	8 byte	no	read, write
I$$	Executable area	4 byte	yes	execute only
C$$	Constant area	8 byte	yes	read only
A$$	Argument lists	4 byte	no	read, write

Each control section is relocatable. The VAX LOADER concatenates the same control section from all compile units.

A sharable control section is one that can be shared between a number of concurrent executable images. THLL control sections that are non-changeable are sharable. This saves disk space and execution time if more than one copy of the same THLL program is executing at the same time. Each execution image has its own O$$ and A$$ control sections, but they share the same I$$ and C$$ control sections. The program must be "installed sharable" in order for sharing to happen.

The A$$ control section contains the argument lists for THLL programs. Argument lists are not constant on the VAX. Dynamic addresses are supplied to an argument list just before calling a procedure. The called procedure copies either the address of the argument (passed by reference) or the value of the argument (passed by value) to the runtime stack frame for that procedure invocation.

B.2 ALLOCATION OF STACK FRAMES

On entry to a procedure, a stack frame is allocated providing memory space for parameters, local variables, and temporaries. On exit from the procedure, the stack frame is released. General register R12 points to the stack frame for the current procedure invocation.

The first 16 THLL words on the stack frame are used to support the THLL procedure environment. This area also serves as a scratch area for THLL macros and library routines. The parameters to a procedure start at offset 16 on the stack frame. Shared variables are next on the stack frame, and temporary variables are last on each stack frame. The cross reference for a procedure indicates the hexadecimal byte offset of all user-defined items on the stack frame.

Stack frames on the VAX are not adjacent. THLL stack frames on the VAX share a stack used to support the normal procedure calling sequence. This stack is used by FORTRAN programs as well as programs written in the MACRO assembly language.

The following diagram describes the structure of the stack frame. The offsets shown are in terms of THLL words.

0	pointer to previous stack frame
1	pointer to argument descriptor list
2	pointer to argument list
3	environment pointer
4 . . 15	temporary locations used by macros and runtime library routines
16 . .	copy or address of actual parameters
. . .	shared variables
. . .	temporary locations used by compiler

B.3 TEMPORARIES

During the compilation process, it sometimes becomes necessary to employ temporary locations on the runtime stack (as shown in the diagram in Section B.2) for storage of intermediate results. If the number of temporaries exceeds the default maximum number, the THLL programmer can increase the maximum number of temporaries via the TMPMAX directive. See the THLL Reference Manual (Reference 1) for more details.

B.4 RUNTIME ERROR MESSAGES

Errors detected by the THLL runtime system cause the following messages to be printed on file SYS$OUTPUT. All messages are followed by an abort of the current image invocation. The following text appears on the same line as the runtime error message:

    AT LOCATION xxxx

where xxxx is a hexadecimal address where the error occurred in the THLL program. The load map indicates which compile unit failed.

1. ARRAY SUBSCRIPT ERROR
   An array subscript value is negative or larger than the declared bound.

2. INVALID DEVICE IN OPEN
   The device number is incorrect. Report this to the THLL group.

3. INVALID FILE POSITION
   The file position given in a random read or write must be greater than 0.

4. INVALID FILENAME IN OPEN
   The filename is invalid for the device code specified.

5. INVALID SIZE IN OPEN
   The file is sequential and the size is less than 1 or greater than 33.

6. MAX REPEAT DEPTH EXCEEDED
   A format repeat nesting level exceeds 10.

7. NO FORMAT GIVEN
   A read or write on a sequential file requires a format statement.

8. POLAR ANGLE OUT OF RANGE
   The angle argument in the POCA function must be in the range -Pi Leq X Leq +Pi.

9. RANDOM FILE REQUIRED
   Filenames INPUT and OUTPUT are illegal for random files.

10. STACK SUBSCRIPT ERROR
    A stack subscript value is not in the range STACKWC(<stack>) Grt X Geq 0.

11. TOO MANY ELEMENTS POPPED FROM STACK
    A POP was attempted on an empty stack.

12. TOO MANY ELEMENTS PUSHED IN STACK
    A PUSH was attempted on a full stack.

APPENDIX C

USING VAX TRICOMP

C.1 INTRODUCTION

VAX TRICOMP translates THLL code into VAX MACRO code. This implies a three-step process in order to create an executable file. These steps are:

1. Compile THLL code producing VAX MACRO code.

2. Assemble VAX MACRO code producing object code.

3. Link the object code into an executable file.

This appendix explains these steps. Additionally, it briefly describes how to produce the various reports available for THLL programs.

Note that filenames under VMS have the format Basename.EXT. Basename is the base filename for a set of related files that have different .EXT's extensions. Most VAX programs have a set of conventions governing the .EXT extensions. This is true for VAX TRICOMP also. The user can override these conventions, but this is discouraged for configuration management reasons. The organization of a THLL program as presented in this appendix is not the only organization, but it is a logical one that should be considered seriously before deviating to another.

C.2 ORGANIZATION OF A THLL PROGRAM

A program written in THLL (targeted for the VAX) consists of one or more compile units, all of which must be compiled, assembled, linked, and then executed. One usually thinks of a compile unit as a module, that is, a set of related procedures and data that can be compiled separately. On the VAX, a single compile unit maps nicely into a single EDT file having a .THL extension. Just as a compile unit maps into a file, a program (which is a collection of compile units) maps into a directory (which is a collection of files). Thus, on the VAX, a program written in THLL consists of one or more .THL files (each of which contains one compile unit) in a common directory. These .THL files are then compiled, assembled, linked, and then executed. Each step of this process creates additional files, most of which have the same base filenames as the .THL files.

C.3 THE VAX TRICOMP ENVIRONMENT

The VAX TRICOMP environment is part of the VAX/VMS environment. It consists of a set of commands to produce an executable THLL program, a global cross reference report, and procedure call tree reports. The set of commands consists of the following:

1. TRICOMP - The command used to compile a THLL compile unit.

2. TMAC - The command used to assemble the VAX MACRO code corresponding to a THLL compile unit.

3. GXRUPDATE - The command used to combine .GXR files into a .MXD file.

4. GXRREPORT - The command used to produce a global cross reference report (.MXR file) from a .MXD file.

5. TREUPDATE - The command used to combine .TRE files into a .MTD file.

6. TREREPORT - The command used to produce a procedure call tree report (.MTR file) from a .MTD file.

7. NTREREPORT - The command used to produce a nested procedure call tree report (.NTR file) from a .MTD file.

These commands are available as all VMS commands are available. The THLL Reference Manual (Reference 1) and the VMS HELP command may be used to receive additional information about the above commands.

C.4 INPUT TO VAX TRICOMP

C.4.1 Compile Units (.THL Files)

By default VAX TRICOMP expects a compile unit in a file with a .THL extension. Each .THL file contains one THLL compile unit. A .THL file is an input source file that can be maintained with a VAX editor. It is recommended that each .THL file have a base filename corresponding to the compile unit name contained in the file.

C.4.2 MAIN Directive

The MAIN directive is used to specify which procedure is the program entry point. When the THLL program is invoked with a RUN command, VMS calls this procedure to start execution on the VAX. This directive must be placed in the compile unit containing the procedure which is being declared the MAIN procedure. There can be only one MAIN directive per program. The MAIN directive has the following format:

```
\ MAIN = procname          or
\\ MAIN = procname
```

By declaring the main procedure to be type INTEGER, the user can control the status returned to VMS. Otherwise, VAX TRICOMP ensures that the main procedure returns a normal status to VMS.

C.4.3 Insert Files (.THI Files)

In addition to the .THL files that correspond to compile units, a THLL program often includes INSERT files. An insert file is an input source file that must have a .THI extension. It can be created simply by editing a file. In order to locate a .THI file associated with the insert declaration:

```
INSERT FILENAME(DIRNAME)
```

VAX TRICOMP searches for DIRNAME in the following order:

1. The VMS logical name table is searched for a definition of DIRNAME. If it exists, that name is the name of the directory that contains the FILENAME.THI file. The DEFINE command in VMS is used to establish this relationship in the logical name table.

2. If DIRNAME is the default identifier, DEFAULT, the current default directory contains the FILENAME.THI file. DEFAULT could be defined in the logical name table in which case the check above would have associated DEFAULT to a (possibly different) directory.

3. Finally, it is assumed that DIRNAME is a subdirectory in the current default directory.

DIRNAME and FILENAME are truncated to 8 characters. This approach gives the program designer quite a bit of flexibility.

C.5 INVOCATION OF VAX TRICOMP (.THL, .THI FILES)

After having created .THL and .THI files, VAX TRICOMP can be invoked by the following command:

```
$ TRICOMP Filename
```

where Filename is the base filename of a .THL file. It is recommended that the current default directory is set to be the directory containing the .THL file before invoking VAX TRICOMP. Otherwise, the association is lost between the .THL file and the files generated by VAX TRICOMP.

C.6 OUTPUT FROM VAX TRICOMP (.TLS, .MAR, .GXR, .TRE FILES)

The two file types, .THL and .THI, described above are created by the users as they develop their program. A .THL file contains a compile unit and it is the input file to a single invocation of VAX TRICOMP. The .THI files contain information to be INSERTed into .THL files during a THLL compilation. VAX TRICOMP compiles a single .THL file containing a single compile unit and produces four files. The default file extensions of the four files created by VAX TRICOMP are:

.TLS - Compiled listing produced by VAX TRICOMP

.MAR - VAX MACRO code file corresponding to the THLL compile unit

.GXR - Global cross reference data for the THLL compile unit

.TRE - Procedure call tree data for the THLL compile unit

By default the base filename of each of these files is that of the .THL file containing the compile unit. VAX TRICOMP creates the four output filenames by appending the above extensions to the base filename of the .THL file. See Section C.15 for ways to override the defaults.

The .TLS files produced by VAX TRICOMP can be examined by the programmers to ensure that their development is proceeding as planned. The .MAR, .GXR, and .TRE files produced by VAX TRICOMP are used as input to other programs.

C.7 INVOCATION OF VAX MACRO (.MAR, .OBJ, .LIS FILES)

The .MAR files produced by VAX TRICOMP contain the VAX MACRO code (Reference 4) corresponding to the THLL compile units contained in the .THL files. These .MAR files must be assembled by the VAX assembler which produces object code and a listing. To invoke MACRO to properly assemble a .MAR file created by VAX TRICOMP, use the following command:

```
$ TMAC Filename
```

where Filename is the base filename of a .MAR file created by VAX TRICOMP. The default file extensions of the two output files created by MACRO are:

.LIS - Assembled listing produced by MACRO

.OBJ - Object code produced by MACRO

Again, the base filenames of these two files are the same as the original .THL file.

The .LIS files are the assembler equivalent of the .TLS files. A THLL user ordinarily does not consult these .LIS files, but they are invaluable when searching for an extremely difficult bug. This is discussed further in DEBUGGING.

C.8 INVOCATION OF VAX LINKER (.OBJ, .MAP, .EXE FILES AND THL$LIBRARY)

The .OBJ files created by MACRO contain VAX object code corresponding to the THLL compile units contained in the .THL files. The .OBJ files are input to the VAX linker, LINK, which collects them into an executable file. Using the VAX linker has many variations. All (or part) of the .OBJ files can be combined into an object library (via the LIBRARY command) and the linker can select them from there, or the linker can simply link the .OBJ files. The details of using the VAX linker can be found in the VAX-11 Linker Reference Manual (Reference 5).

THLL runtime routines must be available to the VAX linker in order for it to properly create a THLL executable program. This is done by defining the logical LNK$LIBRARY to be THL$LIBRARY.

The following is an example invocation of the linker in which several .OBJ files (FILE1, FILE2, FILE3) are collected into an executable file.

```
$ DEFINE LNK$LIBRARY THL$LIBRARY
$ LINK/MAP=OUT/EXECUTABLE=OUT FILE1,FILE2,FILE3
```

This example selects the base filename of the executable file and the map file to be OUT. Thus, the output of this command is two files with the base filename OUT. The default file extensions of these two files are:

.EXE - This file is a VAX executable file

.MAP - This file contains the program's load map

Note that up until this .EXE file, seven files have accumulated for each compile unit. For each Basename.THL file containing a compile unit, the THLL user derives Basename.TLS, Basename.MAR, Basename.GXR, Basename.TRE, Basename.LIS, and Basename.OBJ. The ultimate goal is to combine a collection of .THL files into one executable file. Thus, even starting with three .THL files in a directory, only one .EXE file results (in the same directory).

C.9 RUNNING A PROGRAM (.EXE FILES)

After having created the .EXE file, the program is executed using the following command:

```
$ RUN Filename
```

where Filename is the base filename of the .EXE file.

C.10 DEBUGGING (.LIS FILES)

The art of debugging is not explained here; however, a few tips are listed. The VAX provides a symbolic stack dump whenever a program aborts. This dump is essentially the procedure call stack unwound in a readable format. Since THLL programs adhere to the VAX calling convention, the VAX provided symbolic stack dump can be used to determine the THLL line that aborted. The following is an example of a VAX symbolic stack dump.

```
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name     routine name          line      rel PC    abs PC

DEMOOPER                                        0000014E  00022A1D
DEMOMAIN                                        00000123  00022F32
```

To trace the above dump back to a THLL line number, observe the first module name listed. (Note this module name is the name of the compile unit contained in the .THL file. For the rest of the discussion, it is assumed that the compile unit name is the same as the .THL base filename. This assumption allows concatenation of .LIS and .TLS onto the module name (e.g., base filename) in order to map this module name into files.) In this case the first module is DEMOOPER. The DEMOOPER.LIS file is then edited to search for the string 14E. Note that this is the relative PC at which the program aborted. Macro code can be found to the right of this PC. A line number in the form:

```
; LINE "number"
```

can be found to the right of the macro code. The "number" is the line number in the far left column of the DEMOOPER.TLS file. Finally, examine the .TLS file to determine where the program aborted.

In addition to the symbolic stack dump, the VAX also provides a symbolic DEBUGGER, which allows stepping through a program, and examining registers and memory. A /DEBUG option can be used on the RUN command to enable the symbolic DEBUGGER in the absence of debug symbol tables. The assembly program's symbols are made available to the debugger by using the /DEBUG option when invoking MACRO and LINKER. In this case, the /DEBUG option on the RUN command is not needed to enter the symbolic DEBUGGER when invoking the .EXE file.

C.11 PRODUCING A GLOBAL CROSS REFERENCE REPORT (.GXR, .MXD, .MXR FILES)

The global cross reference report presents an overview of the global symbols of a program. The actual individual compile unit line references for those symbols (as well as symbols local to a compile unit) can be found in the local cross reference at the end of the .TLS file. The global cross reference report is useful in determining which procedures and modules reference a particular global symbol. Symbols defined within an insert file are in a sense global; thus they are also included in the global cross reference report. The global cross reference report is developed from .GXR files in a two-step process.

The first step is to combine the .GXR files into a master cross reference data file (.MXD). This is done by invoking GXRUPDATE. The details of GXRUPDATE are presented in the THLL Reference Manual (Reference 1). The following command invokes GXRUPDATE:

```
$ GXRUPDATE Filename
```

where Filename is the base filename of the desired .MXD file. GXRUPDATE combines the .GXR; files into the Filename.MXD file. The Filename.MXD file is the input file to the second step of this process.

The second step is to transform the .MXD file into a readable global cross reference report. This is done by invoking GXRREPORT. The details of GXRREPORT are presented in the THLL Reference Manual. The following command invokes GXRREPORT:

```
$ GXRREPORT Filename
```

where Filename is the base filename of the .MXD file. The file produced by GXRREPORT is Filename.MXR. This report can be examined and/or printed.

This two-step process allows for incremental updates to the global cross reference data. As the need arises to change selected compile units, the .THL files are changed, compiled using VAX TRICOMP, assembled, and linked. Later, the global cross reference data files are gathered by invoking GXRUPDATE to do a partial update to the .MXD file. GXRREPORT is used to form the new report. The changed compile units are reflected accordingly in the new report.

C.12 PRODUCING A PROCEDURE CALL TREE REPORT (.TRE, .MTD, .MTR FILES)

The procedure call tree report shows which procedures call which procedures. This report is produced in a two-step process that mirrors the two-step process for producing the global cross reference report.

The first step is to combine the .TRE files into a master procedure call tree data file (.MTD). This is done by invoking TREUPDATE. The details of TREUPDATE are presented in the THLL Reference Manual. The following command invokes TREUPDATE:

$ TREUPDATE Filename

where Filename is the base filename of the desired .MTD file. TREUPDATE combines the .TRE; files into the Filename.MTD file. The Filename.MTD file is the input file to the second step of this process.

The second step is to transform the .MTD file into a readable procedure call tree report. This is done by invoking TREREPORT. The details of TREREPORT are presented in the THLL Reference Manual. The following command invokes TREREPORT:

$ TREREPORT Filename

where Filename is the base filename of the .MTD file. The file produced by TREREPORT is Filename.MTR. This report can be examined and/or printed.

This two-step process allows for incremental updates to the procedure call tree data. As the need arises to change selected compile units, the .THL files are changed, compiled using VAX TRICOMP, assembled, and linked. Later, the procedure call tree data files are gathered by invoking TREUPDATE to do a partial update to the .MTD file. TREREPORT is used to form the new report. The changed compile units are reflected accordingly in the new report.

C.13 PRODUCING A NESTED PROCEDURE CALL TREE REPORT (.TRE, .MTD, .NTR FILES)

The nested procedure call tree report shows which procedures call which procedures in a nested format. This report is produced in a two-step process that mirrors the two-step process for producing the regular procedure call tree report.

The first step is exactly the same as the first step of producing a procedure call tree report. The first step is to combine the .TRE files into a master procedure call tree data file (.MTD). This is done by invoking TREUPDATE as shown above. If the .MTD file has been created for a regular procedure call tree report, it can be used for a nested procedure call tree report.

The second step is to transform the .MTD file into a readable nested procedure call tree report. This is done by invoking NTREREPORT. The details of NTREREPORT are presented in the THLL Reference Manual. The following command invokes NTREREPORT:

$ NTREREPORT Filename

where Filename is the base filename of the .MTD file. The file produced by NTREREPORT is Filename.NTR. This report can be examined and/or printed.

As in producing a regular procedure call tree report, this two-step process allows for incremental updates to the nested procedure call tree data.

C.14 DIAGRAM OF A THLL PROGRAM'S FILE RELATIONSHIPS

The preceding description is summarized pictorially in the following diagram. Flow is downward.

```
                        .THL <--- .THI
                          |
                       tricomp ---> .TLS
                       /  |  \
        +-------------+   |   +-------------------+
        |                 |                       |
      .GXR              .MAR                    .TRE
        |                 |                       |
    gxrupdate           tmac ---> .LIS        treupdate
        |                 |                       |
      .MXD              .OBJ                    .MTD
        |                 |                      / \
    gxrreport           link  ---> .MAP   trereport  ntrereport
        |                 |                   |           |
      .MXR              .EXE                .MTR        .NTR
```

C.15 ADVANCED INVOCATIONS OF TRICOMP (OVERRIDING DEFAULTS)

The preceding sections of this appendix describe how to use VAX TRICOMP, accepting the defaults. Sometimes defaults are not sufficient and sometimes defaults constrict a programmer's creativity. If the need arises, VAX TRICOMP provides, in the form command qualifiers, ways to override the defaults. All command qualifiers can be seen in the THLL Reference Manual or via the HELP TRICOMP command.

C.15.1 Suppressing TRICOMP Generated Files

Often utility programs consist of a small number (sometimes one) of compile units. In these cases, the .GXR and .TRE files are cumbersome (or not necessary). The following command does not produce a .GXR and a .TRE file.

```
$ TRICOMP/NOGXR/NOTRE UTIL
```

C.15.2 Parallel Directory Organization

Sometimes a program is so large that keeping all of the files in the same directory would be unmanageable. A valid approach is to view a program as a collection of parallel directories where:

1. [PROGRAM.THL] contains the .THL files,

2. [PROGRAM.TLS] contains the .TLS files,

3. [PROGRAM.MAR] contains the .MAR files,

4. [PROGRAM.GXR] contains the .GXR files, and

5. [PROGRAM.TRE] contains the .TRE files.

The following command sprays the .TLS, .MAR, .GXR, and .TRE files into the appropriate directories for [PROGRAM.THL]FILE1.THL.

```
$ TRICOMP/TLS=[PROGRAM.TLS]/MAR=[PROGRAM.MAR]-
$_/GXR=[PROGRAM.GXR]/TRE=[PROGRAM.TRE]  [PROGRAM.THL]FILE1
```

C.15.3 Overriding Default File Extensions

Occasionally programmers believe that their own conventions are more expressive than the default conventions. The following command compiles a THLL compile unit contained in file PROG.SRC and places the MACRO ASSEMBLER code in file PROG.ASM.

```
$ TRICOMP/NOTRE/NOGXR/MAR=.ASM  PROG.SRC
```

Note that the THLL listing is placed in the default PROG.TLS file.

C.15.4 Example Compile and Assemble Command Procedure

VAX TRICOMP produces VAX MACRO code which must be assembled by the VAX MACRO Assembler. Many programmers are not interested in the VAX MACRO code as they just want the .OBJ files which can be linked to form the .EXE file. Thus many programmers consider this two step process of compiling and assembling as one, where VAX TRICOMP produces a .OBJ file from a .THL file. If there are no compile errors in the .THL file, the following command procedure produces a .OBJ file from a .THL file.

```
$ ON WARNING THEN EXIT
$ TRICOMP/NOGXR/NOTRE 'P1'
$ MNAME = FSPARSE(P1,,,"NAME")
$ TMAC 'MNAME'
```

```
$ DEL 'MNAME'.LIS;
$ DEL 'MNAME'.MAR;
$ EXIT
```

C.15.5 $SEVERITY Returned from VAX TRICOMP

The preceding example command file relies upon the $SEVERITY system symbol. When TRICOMP exits to VMS, $SEVERITY is set to indicate how the compilation went. The following values can be found in $SEVERITY.

1. $SEVERITY = 1 implies a normal compilation

2. $SEVERITY = 0 implies normal compilation with compile errors

3. $SEVERITY = 2 implies abnormal compilation

4. $SEVERITY = 4 implies one of the pertinent files could not be found or opened

Note that if the \\ ABORT directive is contained within a compile unit, then a $SEVERITY of 0 becomes a 2 and a $SEVERITY of 2 becomes a 4.

C.16 EXAMPLE SESSION ON USING VAX TRICOMP

The following is an example session using the VAX TRICOMP environment. All of the features discussed above are covered in the example. Users confused by the above descriptions may want to create the necessary files and actually perform this session. The necessary files are DEMOMAIN.THL, DEMOOPER.THL, and EXTEND.THI. These files are listed in this appendix following this example. An understanding of VAX facilities for creating files is assumed. A slight twist on the example is to try putting the EXTEND.THI file in the same (or parallel) directory as the .THL files.

```
$ SD .

  UDISK1:[username]

$ CREATE/DIR [.DEMO]
$ SD.DEMO

  UDISK1:[username.DEMO]

$
$ ! create DEMOMAIN.THL and DEMOOPER.THL (listed in appendix)
$ ! using one of the VAX editors.
$
```

```
$ DIR

Directory UDISK1:[username.DEMO]

DEMOMAIN.THL;1   DEMOOPER.THL;1

Total of 2 files.
$ CREATE/DIR [.INSERTS]
$ SD.INSERTS

  UDISK1:[username.DEMO.INSERTS]

$
$ ! create EXTEND.THI (listed in appendix) using one of the
$ ! VAX editors
$
$ SD . !sets home directory as default directory

  UDISK1:[username]

$ SD.DEMO ! directory containing the DEMO program

  UDISK1:[username.DEMO]

$
$ ! show the program's structure
$
$ DIR

Directory UDISK1:[username.DEMO]

DEMOMAIN.THL;1   DEMOOPER.THL;1   INSERTS.DIR;1

Total of 3 files.
$ SD.INSERTS ! subdirectory containing insert files

  UDISK1:[username.DEMO.INSERTS]

$ DIR

Directory UDISK1:[username.DEMO.INSERTS]

EXTEND.THI;1

Total of 1 file.
$ SD <

  UDISK1:[username.DEMO]

$
$ ! compile .THL files (each has one compile unit)
$
```

```
$ TRICOMP DEMOMAIN
 BEGIN VAX TRICOMP
 NORMAL COMPILATION  DEMOMAIN
 END VAX TRICOMP
$ TRICOMP DEMOOPER
 BEGIN VAX TRICOMP
 NORMAL COMPILATION  DEMOOPER
 END VAX TRICOMP
$ DIR

Directory UDISK1:[username.DEMO]

DEMOMAIN.GXR;1    DEMOMAIN.MAR;1    DEMOMAIN.THL;1    DEMOMAIN.TLS;1
DEMOMAIN.TRE;1    DEMOOPER.GXR;1    DEMOOPER.MAR;1    DEMOOPER.THL;1
DEMOOPER.TLS;1    DEMOOPER.TRE;1    INSERTS.DIR;1

Total of 11 files.
$
$ ! assemble the .MAR files generated by VAX TRICOMP
$
$ TMAC DEMOMAIN
$ TMAC DEMOOPER
$ DIR

Directory UDISK1:[username.DEMO]

DEMOMAIN.GXR;1    DEMOMAIN.LIS;1    DEMOMAIN.MAR;1    DEMOMAIN.OBJ;1
DEMOMAIN.THL;1    DEMOMAIN.TLS;1    DEMOMAIN.TRE;1    DEMOOPER.GXR;1
DEMOOPER.LIS;1    DEMOOPER.MAR;1    DEMOOPER.OBJ;1    DEMOOPER.THL;1
DEMOOPER.TLS;1    DEMOOPER.TRE;1    INSERTS.DIR;1

Total of 15 files.
$
$ ! link the .OBJ files into a .EXE file
$
$ DEFINE LNK$LIBRARY THL$LIBRARY
$ LINK/MAP=DEMO/EXECUTABLE=DEMO DEMOMAIN,DEMOOPER
$ DIR

Directory UDISK1:[username.DEMO]

DEMO.EXE;1        DEMO.MAP          DEMOMAIN.GXR;1    DEMOMAIN.LIS;1
DEMOMAIN.MAR;1    DEMOMAIN.OBJ;1    DEMOMAIN.THL;1    DEMOMAIN.TLS;1
DEMOMAIN.TRE;1    DEMOOPER.GXR;1    DEMOOPER.LIS;1    DEMOOPER.MAR;1
DEMOOPER.OBJ;1    DEMOOPER.THL;1    DEMOOPER.TLS;1    DEMOOPER.TRE;1
INSERTS.DIR;1

Total of 17 files.
$
$ ! run the DEMO program
$
$ RUN DEMO
```

```
ENTER OPERATION(+,-,*,/,E):
+
ENTER FIRST OPERATOR:
2.4
ENTER SECOND OPERATOR:
3.0
THE RESULT IS:     5.40000
ENTER OPERATION(+,-,*,/,E):
BAD
INVALID CHOICE, TRY AGAIN.
ENTER OPERATION(+,-,*,/,E):
/
ENTER FIRST OPERATOR:
21
ENTER SECOND OPERATOR:
3
THE RESULT IS:     7.00000
ENTER OPERATION(+,-,*,/,E):
E
$
$ ! Produce a Global Cross Reference Report
$
$ GXRUPDATE DEMO
$ DEL *.GXR;        ! Not needed after the update
$ DIR

Directory UDISK1:[DEMO]

DEMO.EXE;1        DEMO.MAP;1        DEMO.MXD;1        DEMOMAIN.LIS;1
DEMOMAIN.MAR;1    DEMOMAIN.OBJ;1    DEMOMAIN.THL;1    DEMOMAIN.TLS;1
DEMOMAIN.TRE;1    DEMOOPER.LIS;1    DEMOOPER.MAR;1    DEMOOPER.OBJ;1
DEMOOPER.THL;1    DEMOOPER.TLS;1    DEMOOPER.TRE;1    INSERTS.DIR;1

Total of 16 files.
$ GXRREPORT/TITLE="Demonstration Use of TRICOMP on VAX"
$ DIR

Directory UDISK1:[username.DEMO]

DEMO.EXE;1        DEMO.MAP;1        DEMO.MXD;1        DEMO.MXR;1
DEMOMAIN.LIS;1    DEMOMAIN.MAR;1    DEMOMAIN.OBJ;1    DEMOMAIN.THL;1
DEMOMAIN.TLS;1    DEMOMAIN.TRE;1    DEMOOPER.LIS;1    DEMOOPER.MAR;1
DEMOOPER.OBJ;1    DEMOOPER.THL;1    DEMOOPER.TLS;1    DEMOOPER.TRE;1
INSERTS.DIR;1

Total of 17 files.
$
$ ! Produce a Procedure Call Tree Report
$
$ TREUPDATE DEMO
$ DEL *.TRE;        ! Not needed after the update
$ DIR
```

```
Directory UDISK1:[username.DEMO]

DEMO.EXE;1        DEMO.MAP;1        DEMO.MTD;1        DEMO.MXD;1
DEMO.MXR;         DEMOMAIN.LIS;1    DEMOMAIN.MAR;1    DEMOMAIN.OBJ;1
DEMOMAIN.THL;1    DEMOMAIN.TLS;1    DEMOOPER.LIS;1    DEMOOPER.MAR;1
DEMOOPER.OBJ;1    DEMOOPER.THL;1    DEMOOPER.TLS;1    INSERTS.DIR;1

Total of 16 files.
$ TREREPORT/TITLE="Demonstration Use of TRICOMP on VAX" DEMO
$ DIR

Directory UDISK1:[username.DEMO]

DEMO.EXE;1        DEMO.MAP;1        DEMO.MTD;1        DEMO.MTR;1
DEMO.MXD;         DEMO.MXR;         DEMOMAIN.LIS;1    DEMOMAIN.MAR;1
DEMOMAIN.OBJ;1    DEMOMAIN.THL;1    DEMOMAIN.TLS;1    DEMOOPER.LIS;1
DEMOOPER.MAR;1    DEMOOPER.OBJ;1    DEMOOPER.THL;1    DEMOOPER.TLS;1
INSERTS.DIR;1

Total of 17 files.
$
$ ! Produce a Nested Procedure Call Tree Report
$ ! Note the .MTD file already exists
$
$ NTREREPORT/TITLE="Demonstration Use of TRICOMP on VAX" DEMO
$ DIR

Directory UDISK1:[username.DEMO]

DEMO.EXE;1        DEMO.MAP;1        DEMO.MTD;1        DEMO.MTR;1
DEMO.MXD;         DEMO.MXR;         DEMO.NTR;         DEMOMAIN.LIS;1
DEMOMAIN.MAR;1    DEMOMAIN.OBJ;1    DEMOMAIN.THL;1    DEMOMAIN.TLS;1
DEMOOPER.LIS;1    DEMOOPER.MAR;1    DEMOOPER.OBJ;1    DEMOOPER.THL;1
DEMOOPER.TLS;1    INSERTS.DIR;1

Total of 18 files.
```

C.16.1 Example THLL Compile Unit (DEMOMAIN.THL FILE)

```
DEMOMAIN
     BEGIN
     \\ MAIN = MAIN
     INSERT EXTEND(INSERTS) ;
     EXTERNAL PROCEDURE ADD ;
     EXTERNAL PROCEDURE DIVIDE ;
     EXTERNAL PROCEDURE MULTIPLY ;
     EXTERNAL PROCEDURE SUBTRACT ;
     DEVICE PRT = SPRINT ;
     DEVICE KB = KBDSS ;
     FORMAT F1(A'28') ;
     FORMAT F2(A'1') ;
     GLOBAL PKB,PPRT ;
     OWN POINTER PKB,PPRT ;
     OWN ALPHA INALPHA(0),PRMTINIT(27),PRMTERR(27) ;
     PRESET
          BEGIN
          PRMTINIT = #' ENTER OPERATION(+,-,*,/,E):' ;
          PRMTERR = #' INVALID CHOICE, TRY AGAIN. ' ;
          END ;
     OWN INTEGER ARRAY DUMMY(32) ;
     GLOBAL MAIN ;
     DEFINE PROCEDURE MAIN ;
          BEGIN
          PKB = OPEN(KB,#'',20) ;
          PPRT = OPEN(PRT,#'',20) ;
          WHILE TRUE DO
               BEGIN
               WRITE(PPRT,LOC DUMMY(0),0,F1,0,PRMTINIT) ;
               READ(PKB,LOC DUMMY(0),0,F2,0,INALPHA) ;
               IF INALPHA EQL #'+' THEN
                    ADD
               ELSEIF INALPHA EQL #'*' THEN
                    MULTIPLY
               ELSEIF INALPHA EQL #'-' THEN
                    SUBTRACT
               ELSEIF INALPHA EQL #'/' THEN
                    DIVIDE
               ELSEIF INALPHA EQL #'E' THEN
                    LOOPEXIT
               ELSE
                    WRITE(PPRT,LOC DUMMY(0),0,F1,0,PRMTERR)
               ENDIF ;
               END
          ENDDO ;
          END ;                  /* MAIN */
     END FINIS
```

C.16.2 Example THLL Compile Unit (DEMOOPER.THL FILE)

```
DEMOOPER
     BEGIN
     INSERT EXTEND(INSERTS) ;
     EXTERNAL POINTER PKB,PPRT ;
     FORMAT F1(A'28') ; FORMAT F2(P'5'F'11') ;
     FORMAT F3(A'16',P'5'F'11') ;
     OWN ALPHA PRMT1ST(27),PRMT2ND(27),PRMTRES(15) ;
     PRESET
          BEGIN
          PRMT1ST = #' ENTER FIRST OPERATOR:        ' ;
          PRMT2ND = #' ENTER SECOND OPERATOR:       ' ;
          PRMTRES = #' THE RESULT IS: ' ;
          END ;
     OWN REAL FIRST,SECOND,RESULT ;
     OWN INTEGER ARRAY DUMMY(32) ;
     GLOBAL ADD,DIVIDE,MULTIPLY,SUBTRACT ;
     DEFINE PROCEDURE READINPUT ;
          BEGIN
          WRITE(PPRT,LOC DUMMY(0),0,F1,0,PRMT1ST) ;
          READ(PKB,LOC DUMMY(0),0,F2,0,FIRST) ;
          WRITE(PPRT,LOC DUMMY(0),0,F1,0,PRMT2ND) ;
          READ(PKB,LOC DUMMY(0),0,F2,0,SECOND) ;
          END ;                  /* READINPUT */
     DEFINE PROCEDURE ADD ;
          BEGIN
          READINPUT ;
          RESULT = FIRST + SECOND ;
          WRITE(PPRT,LOC DUMMY(0),0,F3,0,PRMTRES,RESULT) ;
          END ;                  /* ADD */
     DEFINE PROCEDURE DIVIDE ;
          BEGIN
          READINPUT ;
          RESULT = FIRST / SECOND ;
          WRITE(PPRT,LOC DUMMY(0),0,F3,0,PRMTRES,RESULT) ;
          END ;                  /* DIVIDE */
     DEFINE PROCEDURE MULTIPLY ;
          BEGIN
          READINPUT ;
          RESULT = FIRST * SECOND ;
          WRITE(PPRT,LOC DUMMY(0),0,F3,0,PRMTRES,RESULT) ;
          END ;                  /* MULTIPLY */
     DEFINE PROCEDURE SUBTRACT ;
          BEGIN
          READINPUT ;
          RESULT = FIRST - SECOND ;
          WRITE(PPRT,LOC DUMMY(0),0,F3,0,PRMTRES,RESULT) ;
          END ;                  /* SUBTRACT */
     END FINIS
```

C.16.3 Example THLL Insert File (EXTEND.THI FILE)

```
\\ BOUNDS = 0 , XREF = 3

COMMENT THESE SYNONYM DECLARATIONS ARE USED TO EXTEND THLL ;

SYNONYM
     BEGIN
     NEXTCASE = / , / ;
     ELSEIF = / , / ;
     BOOLEAN = / INTEGER / ;
     ENDDO = / / ;
     END ;
```

C.16.4 Example Global Cross Reference Report (DEMO.MXR FILE)

GLOBAL CROSS REFERENCE Demonstration Use of TRICOMP on VAX   PAGE 1

SYMBOL NAME	ATTRIBUTES	DEFINED IN DECK		USED IN DECK	USED IN PROCEDURE	REFERENCES USED	REFERENCES CHNG
ADD	P(0)	DEMOOPER					
				DEMOMAIN	MAIN	1	
DIVIDE	P(0)	DEMOOPER					
				DEMOMAIN	MAIN	1	
ELSEIF	SYNONYM	EXTEND	I	DEMOMAIN	MAIN	4	
ENDDO	SYNONYM	EXTEND	I	DEMOMAIN	MAIN	1	
MAIN	P(0)	DEMOMAIN					
MULTIPLY	P(0)	DEMOOPER					
				DEMOMAIN	MAIN	1	
OPEN	X			DEMOMAIN	MAIN	2	
PKB	PV	DEMOMAIN		DEMOMAIN	MAIN	1	1
				DEMOOPER	READINPUT	2	
PPRT	PV	DEMOMAIN		DEMOMAIN	MAIN	2	1
				DEMOOPER	ADD	1	
					DIVIDE	1	
					MULTIPLY	1	
					READINPUT	2	
					SUBTRACT	1	
READ	X			DEMOMAIN	MAIN	1	
				DEMOOPER	READINPUT	2	
SUBTRACT	P(0)	DEMOOPER					
				DEMOMAIN	MAIN	1	
WRITE	X			DEMOMAIN	MAIN	2	
				DEMOOPER	ADD	1	
					DIVIDE	1	
					MULTIPLY	1	
					READINPUT	2	
					SUBTRACT	1	

C.16.5 Example Procedure Call Tree Report (DEMO.MTR FILE)

```
PROCEDURE CALL TREE Demonstration Use of TRICOMP on VAX      PAGE   1

        ADD        G          DEMOOPER
                   READINPUT L
                   WRITE      X

        DIVIDE     G          DEMOOPER
                   READINPUT L
                   WRITE      X

        MAIN       G          DEMOMAIN
                   ADD
                   DIVIDE
                   MULTIPLY
                   OPEN       X
                   READ       X
                   SUBTRACT
                   WRITE      X

        MULTIPLY   G          DEMOOPER
                   READINPUT L
                   WRITE      X

        READINPUT             DEMOOPER
                   READ       X
                   WRITE      X

        SUBTRACT   G          DEMOOPER
                   READINPUT L
                   WRITE      X
```

C.16.6 Example Nested Procedure Call Tree Report (DEMO.NTR FILE)

```
PROCEDURE CALL TREE Demonstration Use of TRICOMP on VAX   PAGE    1
MAIN TREE                                         10-JUL-1984 10:18:01


LINE  LEVEL ROUTINE
----  ----- -------

  1     1 MAIN
  2     2    ADD
  3     3       READINPUT
  4     4         -READ
  5     4         -WRITE
  6     3      -WRITE
  7     2    DIVIDE
  8     3       READINPUT      (3)
  9     3      -WRITE
 10     2    MULTIPLY
 11     3       READINPUT      (3)
 12     3      -WRITE
 13     2   -OPEN
 14     2   -READ
 15     2    SUBTRACT
 16     3       READINPUT      (3)
 17     3      -WRITE
 18     2   -WRITE
```

DISTRIBUTION

Library of Congress Attn: Gift and Exchange Division Washington, DC 20540	(4)
General Electric Company Ordnance Systems 100 Plastics Avenue Pittsfield, MA 01201 Attn: G. Desmarais	(6)
J. Fenton	(6)
Strategic Systems Program Office Department of the Navy Washington, DC 20376 Attn: SP-23115 (C. Chappell)	(1)
SP-23423 (H. Cook)	(1)
EG&G Washington Analytical Services Center P.O. Box 552 Dahlgren VA 22448 Attn: IMC	(2)
Local: K50-GE	(1)
K51	(2)
K52	(12)
K53	(50)
K54	(20)
E31 (GIDEP Office)	(1)
E431	(10)

END

FILMED

6-85

DTIC